**LR语法分析器设计文档**

1. **开发环境**

本程序的编程环境为：基于win10-64bit操作系统，wings IDE 5的集成测试环境，语言为python2.7。

1. **任务说明**

对以下文法G进行语法分析。

文法起始符：E

非终结符：E T F

终结符： num ( ) + - \* /

文法G的产生式为：

E -> E + T | E – T | T

T -> T \* F | T / F | F

F -> ( E ) | num

要求对上述文法构造识别该文法所有活前缀的DFA，构造LR分析表，并构造LR预测分析程序。

**文件集合：**

1. grammarLR.py：LR分析程序的脚本
2. get\_FIRST\_SET\_AND\_FOLLOW\_SET.py：本程序衍生出的单独脚本，通用的FIRST集合FOLLOW集求解器。
3. FIRST\_AND\_FOLLOW\_SET.txt：上面脚本生成的两集合，仅供演示通用脚本的效果。
4. grammarInput.txt：程序的文法输入，有一定的格式约定以便于程序扫描，生成程序中约定的python字典格式。
5. tokens.txt：测试串的输入，默认测试串是符合语法规范的。
6. LRResult.txt：LR分析结果。
7. grammar\_test.txt：为get\_FIRST\_SET\_AND\_FOLLOW\_SET脚本提供一个输入接口，可以单独测试给定文法的FIRST集合FOLLOW集求解效果。

在本次实验中，按照以下两个主要步骤进行：

1. **得到文法G的LR分析表和识别所有活前缀的DFA**

首先需要对文法G做增广处理，得到增广文法G’，再根据G’求解其FOLLOW集。这些信息足以使我们着手构造识别所有活前缀的DFA，值得注意的是，在本实验中，构造DFA的过程的同时也在构造LR分析表（分为ACTION表和GOTO表），这是由于二者的本质是相同的，LR分析表实际上就是识别所有活前缀的DFA的表格抽象。

根据一定的分析，可知该文法属于SLR1文法，因此构造LR（0）项目集规范族足矣。

1. **根据LR分析表，构造LR语法分析程序，并输出语法分析结果**

得到了LR分析表之后，构造LR分析程序便十分简单，只涉及栈操作以及查表操作。即给定一个输入串test\_token，给定分析表ACTION\_TABLE和GOTO\_TABLE，即可开始分析过程。分析程序的输出包括了符号栈的变化序列以及输出串的变化序列。

请注意，为简化程序，程序中没有设计具体的错误处理模块，而是采取简单的中断机制来捕获语法错误。程序默认输入的tokens串是有效的且符合文法规范的，程序可以确保在输入串合语法的情况下，输出一个正确、完整的LL1分析过程。

1. **设计思路**

采取框图的形式说明本实验的主要思路。

程序总体分为四个主要部分，每一部分都有对应的输入输出，并在脚本中已经集成了相应的函数（模块）。

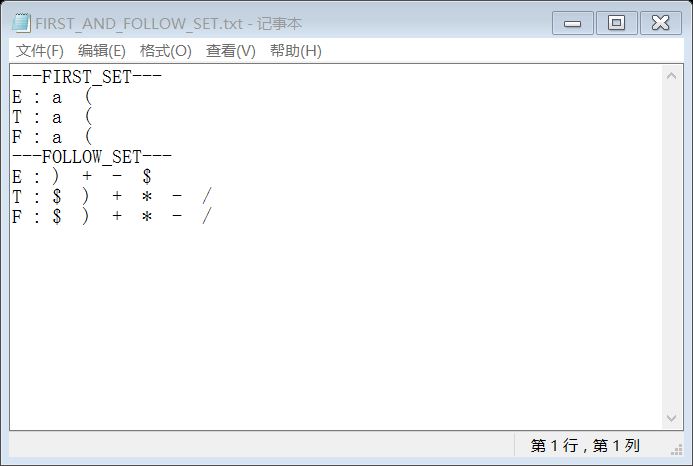
1. **求增广文法G’**

通过过程extendG(P,VN,START\_VN)完成，该过程要求输入一个文法的P,VN和起始符号START\_VN。输出增广之后的文法G’，显然地，G’比G增加了一个产生式，一个VN和改变了START\_VN。该过程输出一个新的文法，包括了新文法的G,VN和P，VT保持不变。

1. **求解FOLLOW集（通用的）**

通过过程getFollowSet (VN,VT,P,FIRST\_SET,START\_VN)完成，该过程要求输入文法的VN,VT,P，由于需要给文法的开始符提供‘$’的初始化支持，因此还需要文法的起始符号。由于求解FOLLOW集一般是基于一个已经给定的FIRST集的，因此我们也将之前求解的FIRST集作为输入。与FIRST求解器相同，该求解器也被设计为通用的，即输入任何一个文法都可以得到相应的FOLLOW集合。

已整合到模块get\_FIRST\_SET\_AND\_FOLLOW\_SET.py中，可以直接使用，该模块要求一个grammar\_test.txt作为文件输入（请注意，grammarInput.txt为LR分析程序的输入接口，而非FIRST/FOLLOW集求解脚本的输入），输出为FIRST\_AND\_FOLLOW\_SET.txt。如本例中的程序，求解两集合的效果如下：



1. **构造识别所有活前缀的DFA并构造LR分析表**

通过过程makeLRTable(P,VN,VT,START\_VN,FOLLOW\_SET)完成，也是本程序的核心部分。该过程要求输入一个文法的全部信息，并返回以下信息。

1. **ACTION\_TABLE**

是LR分析表的ACTIONS部分，存储了规约动作信息和移进动作信息。

1. **GOTO\_TABLE**

是LR分析表的GOTO部分，其实本质是DFA的转移函数的一部分。GOTO表中存储了非终结符的跳转信息。

1. **GO\_FOR\_DFA**

是识别所有活前缀的DFA的转移函数，是DFA结构的一部分。

1. **items**

是LR（0）项目集规范族，也是DFA结构的一部分，它体现为DFA的某一个状态，与转移函数配合则是一个完整的DFA（初始状态为items的第一项）。

1. **构造LR分析程序**

通过过程grammerLR(test\_tokens , ACTION\_TABLE , GOTO\_TABLE)完成，该过程要求利用LR分析表来构造一个完整的LR分析程序，并要求一个输入的token串（在此程序中被简化为python的字符串）。该过程输出两个序列，分别是stack和remains。stack反映了分析过程中符号栈的变化，remains反映了剩余输入串的变化，二者结合即可得到分析过程中的全部信息（移进、规约动作）。

1. **程序实现**
2. **数据结构**
   1. **文法**

本实验用‘#’符号代表‘ε’。

文法的终结符和非终结符均用python的集合表示。

VN = {'E','T','F'}

VT = {'a','+','-','\*','/','(',')'}

文法的产生式用python字典表示，键为非终结符VN，值为产生式右部构成的序列。

P = {

'E' : ['E+T' , 'E-T' , 'T'] ,

'T' : ['T\*F' , 'T/F' , 'F'] ,

'F' : ['(E)' , 'a']

}

简化程序设计，我们默认字符‘a’为代表了num的终结符，我们在测试串中用它来表示任意一个经词法分析器分析后属性为num的token。

* 1. **FIRST集与FOLLOW集**

同样用python字典表示，键为非终结符VN，值为对应VN的FIRST集或FOLLOW集构成的集合。

* 1. **识别所有活前缀的DFA**

分为两个部分。

其一为items，这是一个成员为集合的列表，每个集合都代表了一个LR（0）项目集，集合中的内容为LR（0）项，items构成了LR（0）项目集规范族，它也是DFA的所有状态。默认items的第一项为DFA的初始状态。

其二为GO\_FOR\_DFA，这是DFA的转移函数。

* 1. **LR分析表**

分为两个部分。

其一为ACTION\_TABLE，这是LR分析表中的action部分。其结构为

（状态，vt）：（操作码，操作数）

当操作码为0时，操作数为‘ACC’，表明此时分析成功。

当操作码为1时，操作数为index，表明此时为移进动作，转移状态至index。

当操作码为2时，操作数为产生式form，表明此时为规约动作，利用产生式form规约。

其二为GOTO\_TABLE，这是LR分析表中的goto部分。其结构为

（状态，vn）：index

index为转移之后的状态号。

* 1. **tokens序列**

采用python字符串表示，例如

test\_tokens = '(a+a)\*(a-a)/a'

* 1. **LR分析程序的输出**

输出两个序列，分别为stack和remains。stack反映了分析过程中符号栈的变化，remains反映了剩余输入串的变化，二者结合即可得到分析过程中的全部信息（移进、规约动作）。

1. **模块算法实现**
   1. **求增广文法**

求增广文法极其简单且所有文法通用，仅需要引入一个新的文法起始符S，再添加S->E产生式于文法中即可。

* 1. **FIRST集与FOLLOW集**

由于我们在LL1分析器中已经实现了通用的FIRST集求解器和FOLLOW集求解器，因此可以直接调用。伪代码如下：

*def getFirstSet(VN,VT,P):*

*FIRST\_SET = {}*

*for 任何一个vn:*

*FIRST\_SET[vn] = set()*

*while True:*

*FIRST\_SET\_PRE = FIRST\_SET*

*for 每一个vn的每个产生式:*

*if产生式右侧第一个符号为vt或空:*

*将这个符号加入到vn的FIRST集中*

*else:*

*向后扫描，直到遇到一个不含空的vn为止*

*将这些vn的FIRST集加入到vn的FIRST集中*

*if 一轮扫描后 FIRST\_SET == FIRST\_SET\_PRE:*

*return FIRST\_SET*

比较关键的是何时跳出FIRST求解器。我们需要提前构造一个便于比较的FIRST集合副本FIRST\_SET\_PRE，在每一次扫描完成后，对比当前的FIRST\_SET和副本FIRST\_SET\_PRE是否相同，如果相同表示此时的FIRST集合已经完备，因为它没有在这次扫描中添加任何新的成分。

与FOLLOW集的伪代码如下：

*def getFollowSet(VN,VT,P,FIRST\_SET,START\_VN):*

*FOLLOW\_SET = {}*

*for 任何一个vn:*

*FOLLOW\_SET[vn] = set()*

*添加$符号至START\_VN的FOLLOW集中*

*while True:*

*FOLLOW\_SET\_PRE = FOLLOW\_SET*

*for 每一个vn:*

*for vn的每一个产生式:*

*定位vn在每一个产生式的位置*

*if vn 之后有vt:*

*把这个vt加入到vn的FOLLOW集中*

*elif vn 之后为空:*

*把产生式左部的vn的FOLLOW集*

*并入当前vn的FOLLOW集中*

*else 之后为另一个vn’*

*把vn’的FIRST集并入vn的FOLLOW集中*

*if 一轮扫描后 FOLLOW\_SET\_PRE == FOLLOW\_SET*

*return FOLLOW\_SET*

过程与FIRST集相似，在此不再赘述。

* 1. **Closure算法**

此算法输入一个项目集合I，得到这个项目集合的闭包。伪代码如下：

*def getClosure(I,VN,P):*

*J = I*

*while True:*

*J\_new = J*

*for J\_new中的所有项目:*

*构造集合V，它们是J\_new中每个项目小圆点之后的首字母*

*对于集合V中的vn，*

*把P中所有以vn为左部的产生式加入到J\_new中*

*并置它们为初始项（小圆点位于最左侧）*

*if 一轮扫描中J == J\_new :*

*return J*

这个过程是构造DFA的有力工具。

* 1. **构造识别所有活前缀的DFA和LR分析表**

此过程是本程序的核心，教科书上的算法是先求得DFA，再根据DFA做出LR分析表。但经过简单分析可知，以上两个步骤可以并行执行，因为LR分析表的本质就是DFA的表格抽象。

构造它们的伪代码如下：

*def makeLRTable(P,VN,VT,START\_VN,FOLLOW\_SET):*

*ACTION\_TABLE = {}*

*GOTO\_TABLE = {}*

*GO\_FOR\_DFA = {}*

*items = []*

*构造初始项目集I0并将其置于items的底部*

*镜像一个items\_stack，项目栈中存放未被分析的项目集Ip*

*while True：*

*if项目栈为空：*

*则跳出，结束构造过程*

*弹出项目栈栈顶的项目It*

*项目集Ic = getClosure（It）*

*for Ic中的每一个项目集*

*if 该项目集为新项目集：*

*新项目集入items并压入项目栈中*

*分配给新的项目集index*

*添加该项目集与It的连接信息（DFA的GO和分析表的Sx）*

*if 该项目集中有项目 S->E.*

*填ACC于表中*

*if 该项目集中有除S->E.的规约项目：*

*填Rx于该状态行中*

*return ACTION\_TABLE , GOTO\_TABLE , GO\_FOR\_DFA ,items*

不难看出，分析过程利用一个项目栈，这个项目栈初始化了I0作为栈底，在I0的转移下不断扩增，其实本质上是“深度优先搜索”的过程。当闭包后的项目集是已经分析过的，则舍弃这个项目集。

* 1. **构造LR分析程序**

若已有了分析表信息，分析程序的构造就十分简单，伪代码描述如下：

*def grammerLR(test\_tokens , ACTION\_TABLE , GOTO\_TABLE)：*

*状态0入状态栈*

*test\_tokens + ‘$’为输入串，ip指针指向输入串的首位*

*while True:*

*S是状态栈栈顶，a是ip所指的符号  
 if ACTION\_TABLE[S,a]的种别码是0：*

*分析成功，return*

*elif ACTION\_TABLE[S,a]的种别码是1：*

*S1为种别码后的index*

*把a压入符号栈，把index压入状态栈*

*ip++*

*记录下这次移进操作*

*elif ACTION\_TABLE[S,a]的种别码是2：*

*得到种别码后的规约式form*

*从栈顶弹出form右部，状态栈也弹出相同数量*

*S2是当前栈顶状态，压入状态GOTO\_TABLE[S2,form左部]*

*压入form左部于符号栈中*

*记录符号栈的变化*

*else：*

*报错*

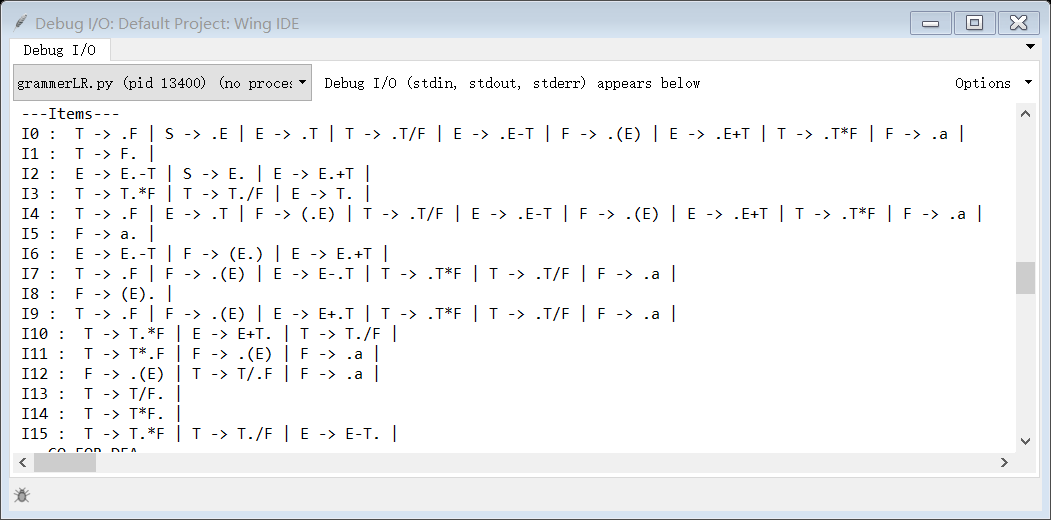
*return 栈和输入串变化信息*

这与书上算法4.3是类似的，过程十分简单，该过程返回栈和输入串的变化信息。

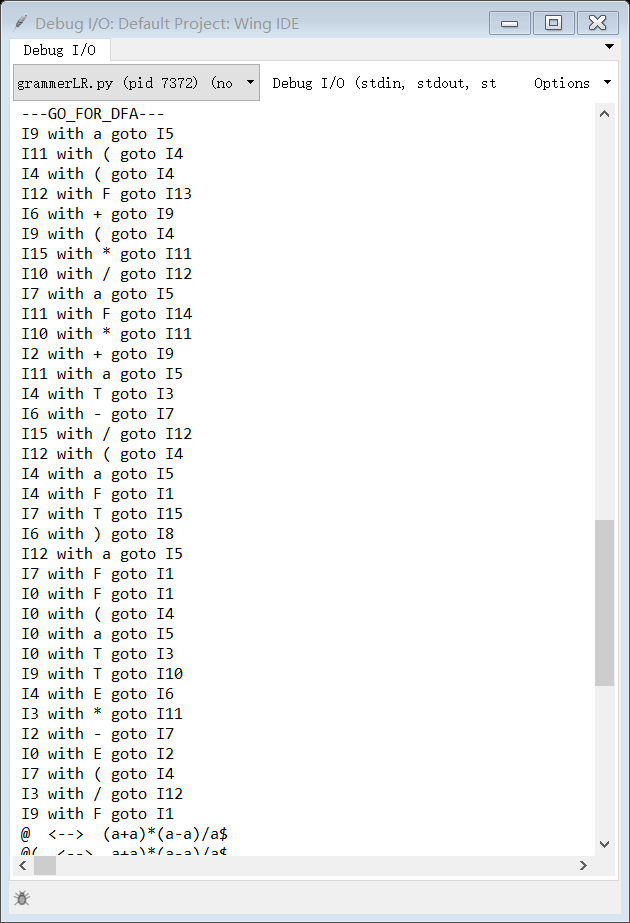
1. **测试用例**

**1、(a+a)\*(a-a)/a**

首先观察输出的DFA，如上面所述，DFA在本程序中被抽象为两个部分，其一是项目集items，其二是转移函数GO\_FOR\_DFA，我们分别打印它们。

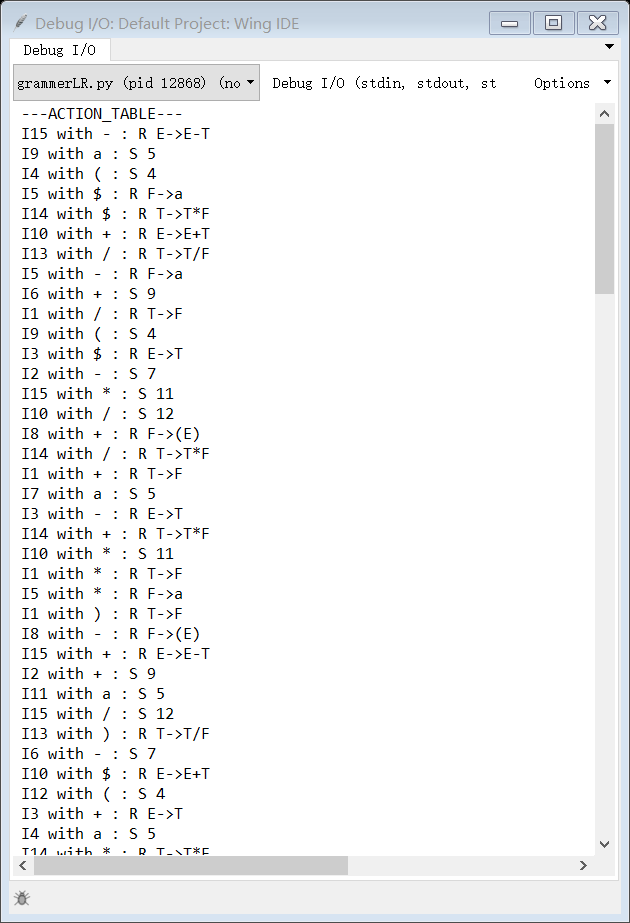


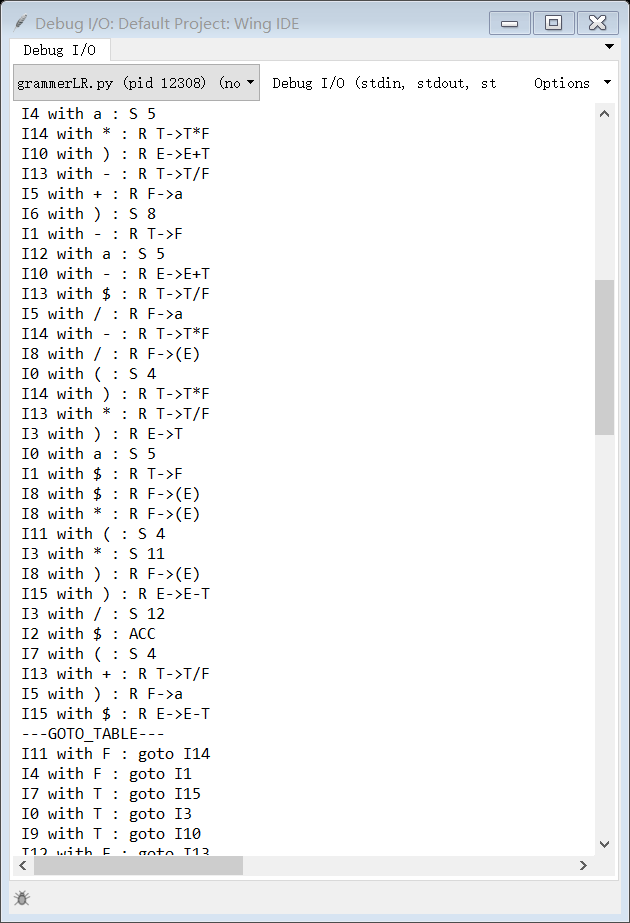
为还原书上的格式，每一行代表了一个项目集，也是DFA的一个状态节点。



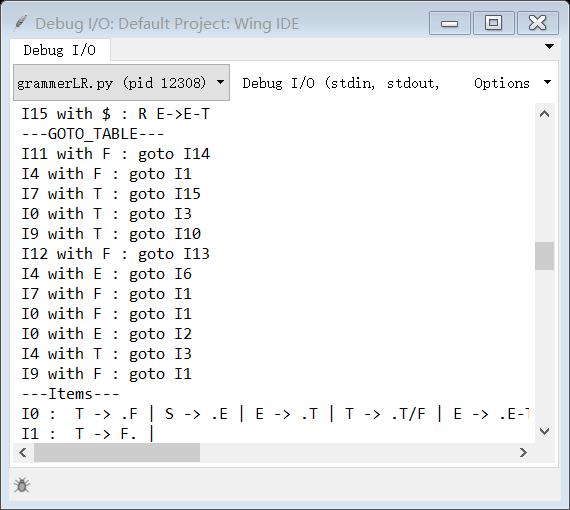
很直观的能看出DFA的各个状态之间的转移和转移条件，值得一提的是由于采用深入优先搜索，转移函数又是一个python集合，因此遍历的时候未必是按照项目集顺序来遍历的。

接着是与DFA同步生成的SLR1分析表，我们仍用字典的方式打印它们。分析表分为两部分，分别是ACTION\_TABLE和GOTO\_TABLE。



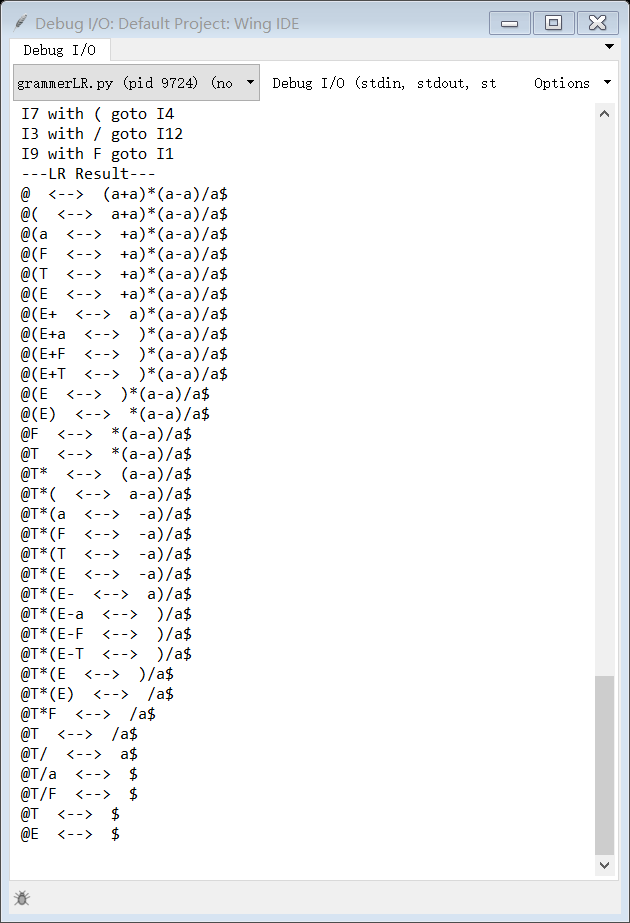


ACTION\_TABLE中我们还原了教材中的分析表格式，将规约动作标注为R，将移进动作标注为S，其中规约动作之后紧随的为本次规约采用的生成式。



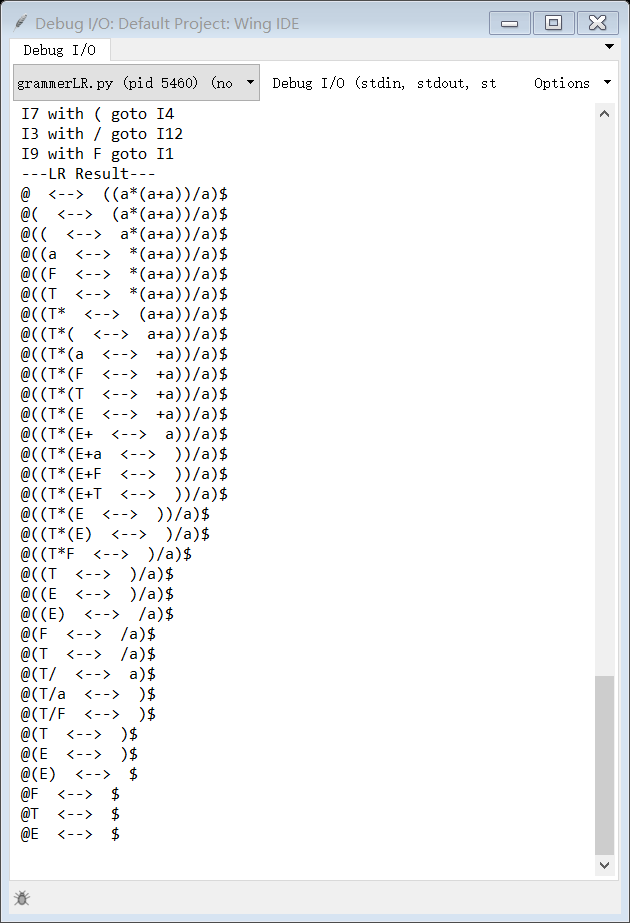
至此我们输出了所有的测试无关项，无论测试用例是什么样子的，DFA和LR分析表应该都是如上的情况。

我们最后输出LR针对测试用例的分析结果



**2、((a\*(a+a))/a)**

由于LR分析表和DFA是测试无关的，我们直接打印LR分析结果。



可以验证上述分析过程是完全正确的。